Matlab 3 Background

General notes

This Matlab assignment is formatted as a function. You write all the code in the sub-functions in the beginning on the file.
Then you test your code by un-commenting the test code in the second half of the file.
Note that you can't put any code before the sub-functions: Matlab won't run it. If your sub-functions need extra definitions (besides the input), define it within the body of the sub-function.
Note: if your console stops being responsive, your code might be stuck in a loop. Hit Ctrl+C repeatedly to get it unstuck.

Problem 1: Binary Division

This problem requires you to implement root finding by binary division.
The input to the problem is a continous function f and an interval $[a, b]$ such that the signs of $f(a)$ and $f(b)$ are different (i.e. one is positive, and other is negative). The intermediate value theorem tells you that there is a value $x \in [a, b]$ such that $f(x) = 0$. We call such value a **root** of f.
The algorithm proceeds by shrinking the interval $[a, b]$ while keeping the root within its bounds.
Specifically, let $a_0 = a$, $b_0 = b$. For $n > 0$, define

$$c_n = (a_n + b_n)/2,$$

so c_n is the midpoint of the segment $[a_{n-1}, b_{n-1}]$.
If $f(c_n) = 0$, we have found the root: $x = c_n$! Otherwise, $f(c_n)$ is either positive or negative, and we define:

$$a_{n+1} = \begin{cases} c_n, \text{ if } a_n \text{ and } c_n \text{ have the same sign} \\ a_n, \text{ otherwise} \end{cases}$$

$$b_{n+1} = \begin{cases} c_n, \text{ if } b_n \text{ and } c_n \text{ have the same sign} \\ b_n, \text{ otherwise} \end{cases}$$

Try out some examples: pick a function, e.g. $f(t) = t^2 - 9$ and consider the intervals $[a, b] = [0, 5]$, and $[a, b] = [-5, 0]$. What are a_1, b_1 going to be in these two cases?
The idea is that the sign of f remains the same at the endpoints of the intervals $[a_n, b_n]$. If it started out as positive at the left endpoint a_0, it will remain positive at a_n for **all** n.

This means that the signs of $f(a_n)$ and $f(b_n)$ are different for all n. (This is called an **invariant** of the algorithm).
This means that the root x is in $[a_n, b_n]$ for all n.
Since $[a_n, b_n]$ gets smaller and smaller (shrinks by a factor of 2 at each iteration), the midpoints c_n of the intervals $[a_n, b_n]$ converge to the root.
Stopping after a finite number of iterations gives an approximation of the root x.
In the problem, you are asked to write code that finds an approximate root x such that $|f(x)| < \epsilon$ for a given ϵ. The pseudocode for the problem is the following:

```
input: f, a, b, epsilon
set c = (a+b)/2
set step_count = 0;
while( |f(c)| > epsilon)
set    a = a_1
   b = b_1
   c = c_1
(according to formulas above).
increase step_count
end
return c, step_count
```

During the execution, the code will loop until a good enough approximation of the root is found. The variables a, b, c will take values a_n, b_n, c_n for $n = 1, 2, \ldots$ - taking the next value with each iteration. The variable $step_count$ will count the total number of times the code looped.

Problem 2: Newton's method

Newton's method exploits the idea that the function is close to its linear approximation. Given $f : \mathbb{R} - > \mathbb{R}$, we start looking for a zero of f (that is, x such that $f(x) = 0$) by choosing a "lucky guess" value x_0. If $f(x_0) = 0$, we are done.
Otherwise, we look at the slope of f at x_0. Intuitively, the slope tells us in which direction to look. If $f(x_0)$ is positive, and $f'(x_0) > 0$, that means that the function is **increasing**, and we should look to the **left** of x_0.
How far should we go? We just look at the tangent line, and go to its x-intercept - call it x_1. The slope of the tangent line (aka linear approximation of f) at x_0 is $f'(x_0)$. Now

$$\text{slope} = \frac{\text{rise}}{\text{run}},$$

and since rise $= f(x_0)$,

$$\text{run} = x_0 - x_1 = \frac{\text{rise}}{\text{slope}} = \frac{f(x_0)}{f'(x_0)}.$$

Therefore, our next value is $x_1 = x_0 - f(x_0)/f'(x_0)$; and in general,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

The pseudocode for the algorithm is the following:

```
input: f, x_0, epsilon
set x = x_0
set step_count = 0;
while( |f(x)| > epsilon)
    set    x = x - f(x)/f'(x)
    increase step_count
end
return x, step_count
```

The variable x will take on values $x_0, x_1, x_2, x_3, \ldots$ as the code runs.
The sequence $\{x_i\}$ will always converge to the root **if** x_0 was a "lucky enough" guess - that is, if it is close enough to the root (or if f is "nice" enough). These terms can be made precise; the set of "lucky guesses" for a root r is called a **basin of attraction** of r (because r "attracts" the iterations that start in the basin). Calculation of the basin of attraction is not trivial; the field of mathematics that studies such questions is called **dynamics**. The set might have complicated properties, e.g. be a fractal.

Problems 1a, 2a: convergence analysis

There is a section that asks you to compare how many steps it takes to reach a given precision with binary division and Newton's method.
The pseudocode is as follows:

```
for n = 1:N
    set epsilon = 1/2^N
    [r, nsteps] = find_zero_with_given_method()
    steps(n) = nsteps
end
plot([1:N], nsteps)
```

Note that in Matlab, you don't need to initialize an array variable to write to it. For large loops, pre-initializing will make code run faster. For instance, this code creates an empty array, fills it with numbers $1^2, 2^2, \ldots, 10^2$, and plots [1..10] vs the squares:

```
sq = zeros(10,1); %10-by-1 array of zeros
for n = 1:10
    sq(n) = n^2
end
plot([1:10], sq)
```

Problem 3: involute

As mentioned in the lab, the involute is a curve that has a an important practical application - in the design of gear teeth.

The goal of this problem is to make you figure out how to combine the techniques of numerical differentiation and integration in one problem.
Suppose you are given a parametrized curve $\gamma(t) = (x(t), y(t))$ and a time t_0. A point on the involute starting from t_0 is the endpoint of a string that has been wound around your curve, and that you start to unwind from $(x(t_0), y(t_0))$.
In particular, at time t_1, this is a point with coordinates $iv(t_1) = (iv_x(t_1), iv_y(t_1)$ such that:

$$\begin{cases} \text{the segment from } \gamma(t_1) \text{ to } iv(t_1) \text{ is } \textbf{tangent} \text{ to the curve } \gamma(t) \text{ at time } t = t_1; \\ \text{the length of this segment is the arclength of } \gamma(t) \text{ from } t = t_0 \text{ to } t = t_1. \end{cases}$$

So, to compute that involute at t_1, you are doing the following:

- compute $L(t_1)$, the arclength of γ from $t = t_0$ to $t = t_1$;
- compute $x'(t)$ and $y'(t)$ at $t = t_1$;
- from $x(t_1), y(t_1)$ move in the direction $(-x'(t_1), -y'(t_1))$ by distance $L(t_1)$.

(Note the negative sign in the direction: we are going backwards because as you move along the curve, you leave the unwound rope behind you).
All these steps are not hard when you do it with discrete approximations. Here's the pseudocode:

```
set t_step = something small
set t = [t_0: t_step : t_max]
set N = length(t)
set L = 0 %this variable will store length of the curve so far
set iv_x, iv_y = zeros(N,1) %N-by-1 arrays
for i = 2:N
    %compute dx, dy
    set dx = x(t(i)) - x(t(i-1))
    %first, update length
    dL = sqrt(dx^2 + dy^2)
    L = L + dL
    %we have all the data now
    %the direction vector is (-dx, -dy), length is L
    %..we can even use the segment function from the previous assignment
    iv_x(i) = x(t(i)) - ...  (figure out what)
    iv_y(i) = y(t(i)) - ...
end
```

Problem 4: extra credit

The idea is that you can calculate the volume of an n-dimensional ball recursively.
We start with dimension 0. The beginning might seem a bit contrived, but see how the pattern holds when the dimension increases.

The 0-dimensional space is a point; the 0-dimensional unit ball is also just one point, so it's zero-dimensional "volume" is $V_0 = 1$.
It doesn't depend on radius (there isn't much space in dimension 0 anyway), so

$$V_0(r) = 1 = 1 \cdot r^0.$$

Let's move on to the next dimension! In dimension 1, the unit ball - the set of points distance at most 1 from the origin - is the interval $[-1, 1]$. You know its length is 2, but it can also be seen as the integral; the cross-sections of the interval are points, that is, zero-dimensional disks, so

$$V_1 = \int_{-1}^{1} V_0 dt = \int_{-1}^{1} 1 dt = 2.$$

The 1-dimensional disk of radius r is just the interval $[-r, r]$, whose length is $2r$. But this can also be seen as a unit disk scaled up by r. In dimension 1, the volume scales linearly, so

$$V_1(r) = V_1 \cdot r = 2r.$$

Things start getting interesting in dimensions 2 and up.
The 2-dimensional unit ball is the unit disk. Its cross-sections perpendicular to the diameter are intervals, that is, one-dimensional disks. The radius at height t is $\sqrt{1-t^2}$, so

$$\begin{aligned} V_2 &= \int_{-1}^{1} V_1(\sqrt{1-t^2}) dt \\ &= \int_{-1}^{1} 2\sqrt{1-t^2}) dt \qquad\qquad = \pi. \end{aligned}$$

The volume of the 2-dimensional ball of radius r is the volume of the unit ball scaled up by r in two dimensions, so

$$V_2(r) = V_2 \cdot r^2 = \pi r^2.$$

Moving on to dimension 3. The unit ball is, well, a ball; its cross-sections perpendicular to the diameter are disks (that is, 2-dimensional balls). The radius at height t is still $\sqrt{1-t^2}$ by the Pythagorean theorem, and so

$$\begin{aligned} V_3 &= \int_{-1}^{1} V_2(\sqrt{1-t^2} dt \\ &= \int_{-1}^{1} \pi(\sqrt{1-t^2}^2 dt \\ &= \int_{-1}^{1} \pi 1 - t^2 dt \\ &= 2\pi(t - t^3/3)\Big|_0^1 \\ &= \frac{4}{3}\pi. \end{aligned}$$

The volume scales proportionately to the cube of the radius in dimension 3, so

$$V_3(r) = V_3 \cdot r^3 = \frac{4}{3}\pi r^3.$$

From here on, the integrals become complicated to do by hand (symbolically). But you can write a MATLAB script to calculate V_n in terms of V_{n-1} (just like we did above). Figure out what the pattern is from the formulas, and write the code.